Danmarks Tekniske Universitet

Introduktion til Softwareteknologi

Gorillasprojekt rapport

Bjarke Bak Jensen
s214957

Hans Henrik Hermansen
s194042

Mathias Vegeberg Edvold
s214973

Jonathan Schmidt Højlev
s194684

Gruppe nr. 8

21. januar 2022

1 Arbejdsfordeling

I løbet af dette projekt har vi alle arbejdet tæt sammen og diskuteret mange idéer fælles for at øge samarbejdet. Derfor er der ikke en klar linje for hvem der har lavet hvad for alting. Dog har vi haft hovedansvarlige på de enkelte dele, og det er dem, der nævnes herunder.

1.1 Rapporten

I rapporten har vi været ansvarlige for følgende afsnit:

- 2: Bjarke
- 3: Bjarke og Jonathan
- 4: Jonathan og Hans
- 5.1-5.4: Hans
- 5.5: Hans, Mathias
- 5.6: Mathias, Jonathan
- 5.8-5.10 Mathias, Bjarke
- 6: Mathias, Jonathan
- 7: Mathias, Bjarke
- 8: Jonathan, Mathias, Bjarke, Hans

1.2 Koden

I programmeringen har vi været ansvarlige for følgende filer:

- Bjarke: Samtlige Images ("sprites"), Building, Platform, Banana, Coconut, Castable.
- Hans: GameObject, MenuController, Level, Map, Helpers, Player, MapObject.
- Mathias: Hele den første prototype, Music, Sound, De forskellige PowerUps.
- Jonathan: Overførsel fra `AdvancedGorillas` til `SimpGorillas`, GameObject, Player, Level,
- Os alle: Grundig testning af programmet, samt generel fejlopsporing i diverse filer.

2 Introduktion

Vi vil i denne rapport beskrive vores proces bag designet og udviklingen af et computerspil baseret på spillet "Gorillas.bas."
Dette spil vil blive skrevet i programmeringssproget Java, hvor vi vil bruge JavaFX til at lave vores grafiske brugergrænseflade. Udviklingen vil ske i to etaper, idet vi først vil lave en simpel version af spillet, som vi kalder SimpGorillas, og så derefter tilføjer udvidelser.

3 Problemanalyse

Opgaven er opdelt i to dele, først den simple del SimpGorillas, og derefter udvidelser til denne, som vi tilsammen kalder AdvancedGorillas. Dette giver også anledning til at analysere og forklare hver dels problemer hver for sig. Dog er der selvfølgelig også noget som begge dele har til fælles, nemlig den grundlæggende idé bag spillet. Spillet er et 2D-spil og består af op til 2 spillere som

kaster projektiler efter hinanden på tur. I det originale spil er spillerne gorillaer og de kaster med bananer. Dette er dog ikke noget krav for opgaven ligesom det også er op til os at vurdere om, og hvordan man vinder spillet.

3.1 SimpGorillas

I SimpGorillas skal der være 2 spillere som er positioneret i hver sit hjørne af skærmen. Størrelsen på denne er $n \times m$ pixels, og derfra skal spillerne skiftes til at kaste "bananer" mod hinanden. Man får et point hvis man rammer inden for en radius på $\frac{n}{50}$ pixler af den anden spiller. Her skal bananen følge en kurve som vi kender det fra fysikkens skrå kast, med en tyngeacceleration på $9,81\frac{m}{s^2}$. Det skal her være muligt for spilleren at vælge størrelsen på skærmen, og at se bananens kurve. Dog skal banen ikke nødvendigvis være animeret.

Her er der allerede flere måder man kan vælge at gøre tingene på. Med hensyn til at vælge størrelsen på skærmen, så kan det enten gøres ved et brugerinput via kommandolinjen, eller via vores grafiske brugergrænseflade. Kommandolinjen er nemmest at bruge under udvikling, da det er simpelt. Men når programmet er færdigt, og det skal køre uafhængigt, som en jar-fil, vil brug af kommandolinje blive en udfordring. Derfor kunne en pænere brugergrænseflade foretrækkes.

Kastet af bananen kan animeres. Dette ville gøre spillet mere interessant for brugeren. Man kunne også have valgt at tegne kurven her. Det kunne have været gjort ved brug af en "PixelWriter"i JavaFX, hvor man kan farve hver eneste pixel i alle de positioner som bananen gennemløber. Dette kan dog ende ud med at se meget firkantet ud, specielt hvis skærmen er for lille. Derudover kan man også tegne banen med ved brug af "QuadCurve-metoden, men denne kræver, at man skal lave flere beregninger frem for de to andre, hvor man bare finder y ud ud fra x og gennemløber alle x-værdier. Da et animeret kast ser bedre ud og ikke er sværere at implementere end de andre løsninger er det en foretrukken løsning.

3.2 AdvancedGorillas

Problemet i `AdvancedGorillas` går ud på at definere en mængde af tilføjelser til `SimpGorillas`. Problemet er dermed ikke klart defineret på forhånd men i stedet noget, vi selv skal tage stilling til og definere. I den forbindelse har vi konkretiseret nogle udvidelser til programmet, som vi har overvejet at tilføje. Herunder analyseres de problemer, der kunne være med dem.

1. Forhindringer i form af bygninger og platforme
2. Mulighed for at starte på forskellige positioner
3. Liv/HP (hit points) til spilleren, således at man dør når man ikke har flere tilbage
4. Man spiller bedst ud af x, hvor x er et ulige tal, og vinder når man har dræbt den anden spiller nok gange til at den anden ikke kan vinde, $\frac{x+1}{2}$ gange.
5. Flere forskellige ting at kaste med, som skader mere.
6. Mulighed for at spilleren kan bevæge sig
7. Power Ups, altså noget som giver spilleren fordele, hvis man får dem.
8. Lyd til spillet
9. Sprites, altså noget grafik som skal afbilde vores elementer i spillet.
10. En Main Menu, hvor man kan ændre spillernes navne, ændre lydniveauet og starte spillet.
11. Bygninger går i stykker når de bliver ramt/hvis de bliver ramt nok gange

Den første udvidelse vil allerede være opnået, hvis vi får skabt nogen objekter som både fungerer som forhindringer for spilleren. Samtidig kan disse forhindringer og bygninger også hjælpe os med udvidelse nr. 2, da spilleren kan placeres på disse objekter. For dette punkt er det ikke nødvendigt at de kan gå i stykker, men det er målet for det sidste punkt. For at de kan gå i stykker, kræver det en særlig mængde af sprites, der kan vise tilstanden af de forskellige objekter.

Hit Points kan tilføjes til spillere, men det kræver dog at vi også definerer skade/"damage" for vores kasteprojektiler. Hvis dette bliver gjort, kan det gøres så generelt som muligt, hvormed det også bliver nemt at tilføje endnu et projektil, hvor skaden er anderledes. Problemet med dette i forhold til spiloplevelse er at det kan blive svært at balancere hvor meget liv henholdsvis skade som spillerne og projektilerne bør have, for at spillet bliver underholdende.

Med mulighed for at bevæge sig, kan man tilføje et nyt element af styring til spillet. På denne måde bliver det nemmere for spilleren at kaste i tilfældet af, at nogle forhindringer er svære at skyde forbi. Desuden gør det spillet mere interessant, idet modstanderen ikke kan blive ved med at skyde samme sted og ramme. Denne bevægelse skal dog være animeret eller på anden måde grafisk illustreret for, at brugeren kan nå at udnytte det. Desuden er der en udfordring i hvordan fysikken bør implementeres, så spillerens bevægelse ser realistisk ud.

Power Ups tænkes som små objekter på skærmen, der giver spilleren en fordel. De har et problem i forhold til, hvordan man skal opnå dem. Her kan det f.eks. vælges mellem om man skal ramme dem med et projektil eller fysisk bevæge sig hen til dem. Vi vil lave dem, så de er ligesom vores forhindringer, men når man opnår en Power Up, så får man adgang til den. Her skal vi desuden tænke over, hvor hyppigt de bliver genereret.

Sprites, grafiske tegninger, kan tilføjes for at gøre spillet pænere at se på. Derfor er den eneste udfordring her blot at vi skal have tegnet vores sprites. Derudover giver dette to muligheder i forhold til hvordan spillet håndterer kollisioner mellem objekter: vi kan enten tjekke om det kastede objekt skærer spilleren grafiske billede, eller vi kan oprette en usynlig "Hit box"for spilleren, altså en ramme for hvornår spilleren tæller som ramt.

3.2.1 Andre mulige udvidelser

Idet denne opgave er meget åben, er der mange mulige udvidelser, der kunne tilføjes. Men på grund af det begrænsede omfang, er der naturligvis nogle vi har valgt fra.

Blandt vores overvejelser har været en gemme-funktion, så man kunne genindlæse et bestemt level i spillet og fortsætte spillet fra hvor man slap. Vi valgte dog at prioritere andre ting såsom grafik, lyd og udvidelser til selve spiloplevelsen, som gjorde spillet mere interessant at spille.

Andre overvejelser har været:

- Endnu flere forskellige Projektiler
- Multiplayer, hvor man kan spille med 3 eller flere gorillaer.
- En singleplayer. Altså mulighed for at spille mod en AI.
- En guide til spillets kontrolmuligheder inde i menuen

Disse udvidelser er ligeledes nedprioriteret i forhold til dem, vi har valgt at implementere, idet vi tog beslutningen baseret på den bedste spiloplevelse.

4 Design

Vi laver vores program efter paradigmet Objektorienteret programmering, og vil bruge konceptet MVC(model-view-controller) til struktureringen af vores program.

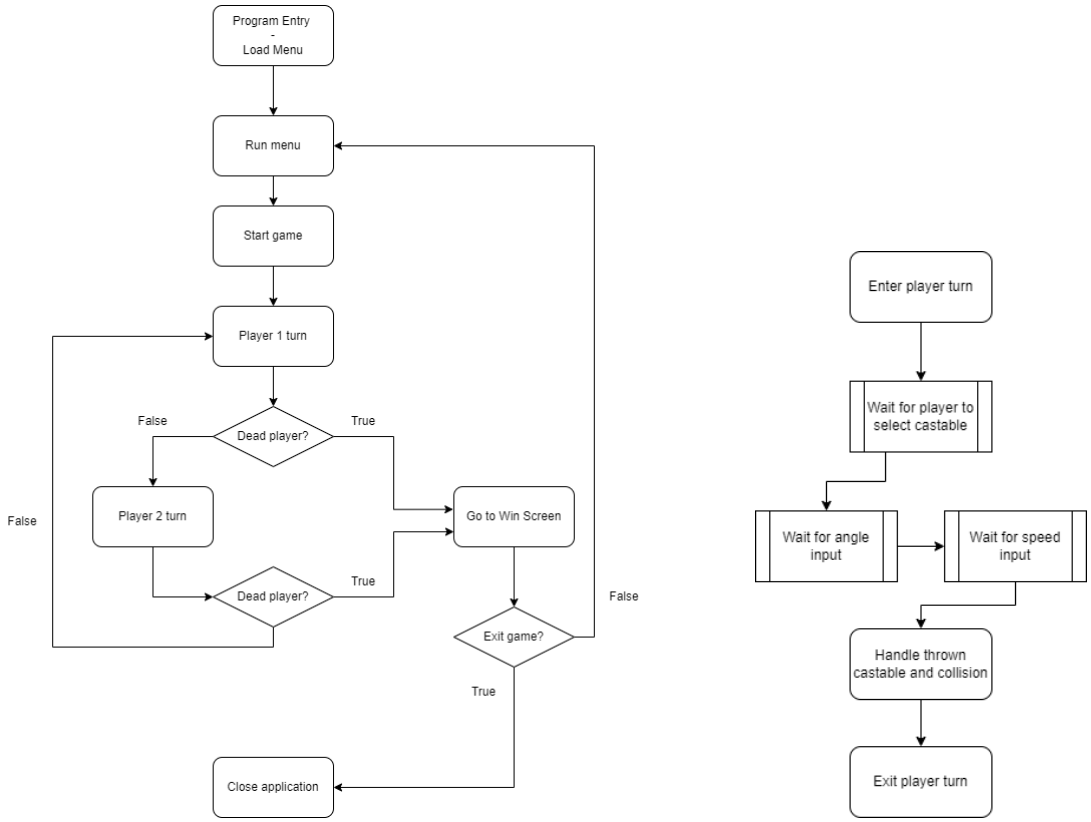
Vores program er et spil, der er opdelt i forskellige scener. Først møder man en menu, og derefter er man i gang med spillet i et "level". Disse dele af vores program kan analyseres i forhold til Model-view-controller konceptet. View-delen handler om hvad brugeren rent faktisk ser fra programmet, model er hvordan objekterne er repræsenterede, og controller er hvordan programmet håndterer begivenhederne fra brugeren.

4.1 Model-View-Controller

De vigtigste elementer at modellere i dette program er spillerne (gorillaerne), kasteobjektet (bananen), samt objekterne på banen. Til at starte med er hele banen modelleret som et koordinatsystem, hvor hver pixel er et punkt. Vi har valgt at designe objekterne todelt, så alle objekter både har en "sprite", altså en visuel del samt en bagvedliggende abstrakt form, der bestemmer hvordan controlleren håndterer objektet. For eksempel vil vi ikke bruge billedet af en banen til at bestemme hvornår bananen er stødt ind i en bygning men i stedet modellere bananen som en cirkel. Ligeledes modellerer vi både gorillaerne og bygningerne som rektangler i koordinatsystemet. Hvis et objekts abstrakte form overlapper med et andets, skal det højst sandsynligt behandles af controlleren. Dette kunne være hvis en spiller rammer en anden spiller med en banan. Dette gør det f.eks. muligt at have komplicerede visuelle "sprites", der i modellen kan behandles som simple figurer, hvilket gør ting som kollisioner mellem objekter nemmere at håndtere.
Denne todeling af objekter gør det muligt at få controlleren til at håndtere spilløkken, inputtet fra brugeren og kommunikation mellem de andre objekter i programmet, uden at controlleren behøver at håndtere meget af model-delen og view-delen uafhængigt af hinanden, da de andre objekter oftest kan håndtere både deres view- og modelrepæsentation på samme tid. F.eks. håndterer spiller-objektet opdatering af sin hitboxs og sprite positioner, når en spiller bevæger sig. Controller har dog altid mulighed for fuld kontrol over både view- og model-delen for alle objekter, og ændrer b.la. ofte på hvilke ting er synlige for spilleren.

4.1.1 Programmets Livcyklus

Vores program/spil følger en livcyklus fra det startes til det bliver lukket af brugeren. Vi designede en simpel version heraf for at give os overblik og en referenceramme senere i implementationsprocessen. I denne har vi også uddybet spillerens tur, da flowet i en spillers tur vil guide vores kommende beslutninger i implementationen. Disse vises i Figur 1.



(a) Program flow (b) Spillers tur

Figur 1: Diagrammer over programmets livcyklus

4.2 Udvidelser og Grafisk design

I den avancerede del vil vi gerne fokusere på at gøre spillet væsentligt mere visuelt interessant og intuitivt at spille. Derfor vil vi også gerne implementere en menu, der giver spilleren mulighed for ændre spillets indstillinger. Denne kan næsten ses som en sekundær controller, der skal påvirke spillets model, men også har meget kontrol over hvad der vises på skærmen f.eks. ved overgang fra menu til spil.
For at forbedre spillerens oplevelse vil vi gerne gøre de muligt at styre vinklen og kraften af kastet med musen og vise spilleren begge dele på en måde, der er nem at forstå. Dette har vi gjort, da vi efter at have set det originale spil, fandt det kedeligt at indtaste værdierne, samtidig med at det også gør spillet lettere, da man derved bare kan finde én kombination af vinkel og starthastighed, der rammer modstanderen og genbruge denne kombination resten af spillet.
Helt konkret har vi designet følgende idé til hvordan denne styring kunne opnås. Når musen bevæges over skærmen, tegnes en linje med fast længde fra den gorilla, hvis tur det er, hen mod musen. Denne linje illustrerer hvilken vinkel spilleren skal til at kaste med. Når musen trykkes, bliver linjen meget kort og begynder at vokse, for at indikere hvilken hastighed man skal til at kaste med. Den stopper, når den når en bestemt længde. Når musen slippes, kaster spilleren det valgte kasteobjekt af sted i den retning og med hastighed svarende til længden af sigtelinjen. Denne linje kommunikerer derfor tydeligt til spilleren hvordan man sigter, skyder og dermed spiller spillet.

Vi vil også implementere forskellige sprites til f.eks. bygningerne og gorillaerne. Dette hjælper

en smule med at gøre spillet mere grafisk interessant, da der derved er noget variation i det man kigger på - spillet bliver mere levende.

5 Implementation

Hele programmet er skrevet i Java, og vi har brugt JavaFX til at implementere den grafiske overflade.

5.1 Generel struktur

Til at starte med har vi inddelt vores program i særskilte klasser ("classes") for at skabe overskuelighed og gøre hele arbejdet nemmere. Desuden hjælper det for at rent praktisk i objektorienterede programmeringssprog som Java, når man kan oprette instanser af diverse klasser. Vi gør også brug af abstrakte klasser, for at samle en del af vore objekter i abstrakte grupper, så vi skaber mulighed for nemmere udvidelser og nemmere håndtering i koden.

5.2 Controller og spilløkken

Vores primære klasse er kaldet `GameObject`, som også er spillets primære controller. Den åbner menuen, starter spillet, og håndterer alle events fra brugeren. Når spillet er i gang kører funktionen `gameLoop`, som giver den aktive spiller en tur og håndterer input fra brugeren (beskrevet under 'design'). Dette objekt er derfor samlingspunkt for hele vores spil og har nem adgang til alle andre objekter i spillet. På den måde er implementationen af spilløkken nemmere, da interaktionen mellem de forskellige objekter i spillet kan håndteres samme sted.

Måden input fra brugeren bliver håndteret på er gennem JavaFX's såkaldte `eventFilter`. Disse kalder en givet funktion, når et specificeret `event opstår`. Dette kunne være når brugeren trykker på musen eller trykker på en tast. Et godt eksempel på dette er vores sigte linje, der gør brug af et eventfilter. Vi opretter et eventfilter, der reagerer på at musen bevæger sig, og giver den vores funktion `drawAimLine`, der tegner vores sigtelinje baseret på musens og spillerens position.

5.3 Menuen

Menuen er implementeret med en controller klasse `MenuController`, der styrer hele menuen. Dette er gjort, da den på mange måder er adskilt fra selve spillet og mere er i samspil med `GameObject`. Denne controller kontrollerer overgangene mellem de forskellige menu-scener og eventuelle ændringer i menuen.
Selvom menuen og spillet på mange måder er adskilte, har menuen og spillet stadig brug for adgang til hinanden, når de skal ændre på elementer i hinanden. Dette kunne f.eks. være når en spillers navn ændres i menuen eller spillet skal ændre titlen på vinder-menuen, når en spiller vinder. Derfor har begge objekter en intern reference til hinanden.
Menuen er opdelt i flere "scener", der svarer til de forskellige undermenuer, der er i menuen f.eks. hovedmenuen (`mainMenu`) eller indstillinger (`optionsMenu`). Disse undermenuer implementeres gennem klassen `Menu`, der er en privat klasse i `menuController`. Denne klasse håndterer det visuelle setup og konstruktion af menuerne. Dette gør det nemt at oprette nye undermenuer, der passer ind i æstetikken, vi har valgt til dem. Hver undermenu har knapper, der enten skifter til en anden undermenu eller påvirker spillet. Disse knapper oprettes nemt gennem `Menu` objektet.

5.4 Animation

Animationerne i vores program bliver lavet ved brug af det som i JavaFX hedder en `TimeLine`. En `TimeLine` er praktisk talt et objekt, der kalder en given funktion en bestem mængde gange med en given forsinkelse mellem hver funktionskald. Disse anvendes bl.a. til animering af en kastet banan. Her er funktionen, der gives, `animateCastible`, der opdater bananens hastighed og position og stopper animationen, når bananen støder ind i noget. For denne `TimeLine` skal funktionen kaldes et ukendt antal gange. Derfor kører vi animationen uendeligt, indtil vi stopper den manuelt. Det er primært vores controller objekt, der står for animationer, da de er en stor del af en enkel gennemgang af spilløkken. `Level` objektet har dog en enkelt animation, der bare er spillets baggrund.

5.5 Level, Map og MapObjects

5.5.1 Level

`Level` objektet er implementeret for at samle alle vores baner (`Map` objekter) og metoder for at opsætte dem i starten af et nyt spil i ét objekt for simplificere vores primære controller en smule.

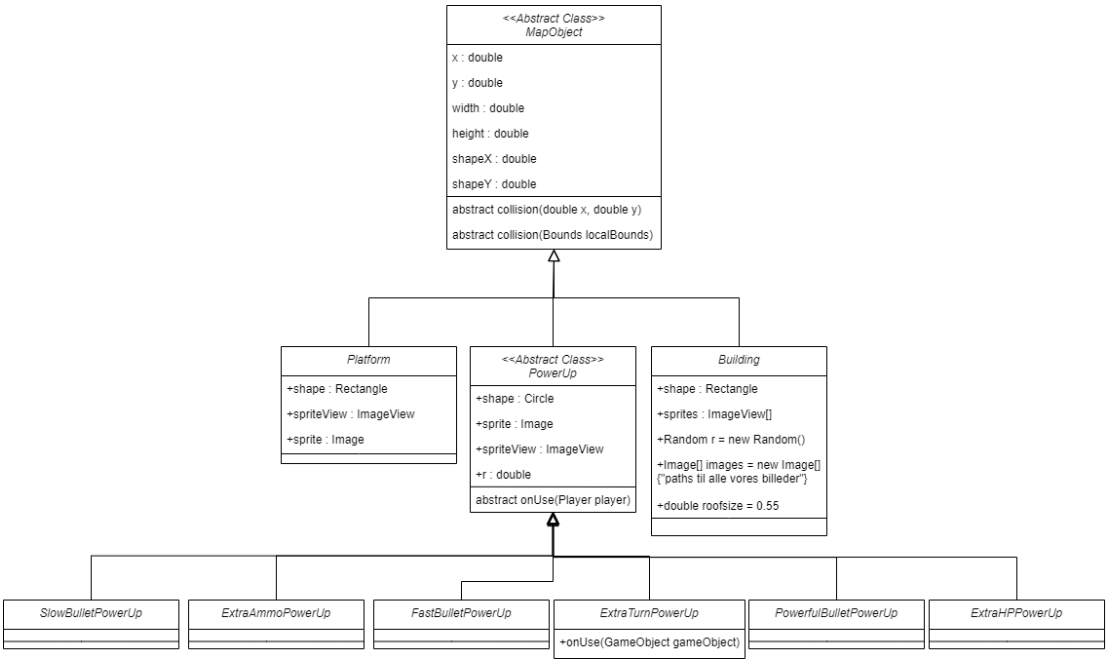
5.5.2 Map

`Map` klassen er meget simpel. Klassen tillader nem oprettelse af nye baner med metoder for nem placering og oprettelse af platforme og bygninger. Den benytter sig således blot af de andre klasser vi har implementeret. Det vigtigste i `Map` klassen er dens `ArrayList`'er af `mapObjects`, `buildings`, `platforms` og `powerUps`. På den måde indeholder et `Map` objekt derfor al informationen hvad der er på en bane, undtagen spillerne selv.

5.5.3 MapObjects

Til at lave vores forhindringer, altså bygninger og platforme, har vi lavet en abstrakt klasse som vi har kaldt `MapObject`, se Figur 2. Deraf har vi to subklasser, som er forhindringer, nemlig `Building` og `Platform`. Disse to har nemlig meget til fælles, f.eks. at de begge skal have en hitbox, som afgør hvornår de bliver ramt, men også en position i spillet. Alligevel er der dog forskel på, hvordan de bliver genereret, da bygningerne skal stå på jorden, mens platformene er oppe i luften. Men det smarteste ved den abstrakte superklasse er, at det er nemt at tilføje nye "map objects,"med alle former og størrelser, også selvom de måske skal have flere egenskaber end de allerede implementerede bygninger og platforme. Hvilket vi netop har gjort med power-ups.

For at lave vores power-ups har vi gjort brug af en abstrakt klasse, `PowerUp`, der nedarver fra `MapObject`. Alle power-ups nedarver fra denne klasse. Klassen indeholder nogle metoder til at oprette og holde styr på en power-up. Derudover indeholder den en abstrakt metode, `onUse`, der kræver, at alle power-up-klasser implementerer denne metode. Selve power-up-klasserne er meget simple, idet de kun indeholder en konstruktør og den førnævnte `onUse`-metode. Det er implementeringen af denne metode, der differentierer de forskellige power-ups. De er all ret simple: For eksempel fordobler `SlowBulletPowerUp`'s `onUse`-metode bare den valgte projektils vægt, mens `ExtraAmmoPowerUp` giver den nuværende spiller en ekstra kokosnød. Der er i alt 6 forskellige power-ups. Alle power-ups bliver styret af `GameObject`-klassen, der en gang imellem laver et nyt og tilfældigt power-up. Man samler et power-up op ved at hoppe hen til den. Derefter får man selv lov til at styre, hvornår man vil bruge et af sine power-ups.



Figur 2: Diagram over nedarvning fra den abstrakte MapObject klasse. Her vist uden getter- og setter-metoder. (Bemærk at teksten i Image[] i Building selvfølgelig har de egentlige stier stående i programmet).

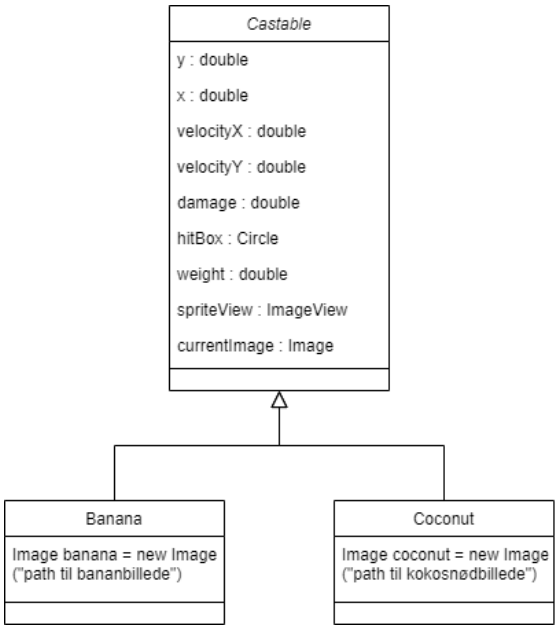
5.6 Spillere og Projektiler

5.6.1 Spillere

Spillerne i `Advanced Gorillas` er implementeret som en selvstændig klasse, hvis objekt indeholder al relevant information om spilleren i løbet af spillet. Spilleren bliver oprettet og opbevaret i `Level` objektet (og ikke `Map` objektet). Det er i `Player` klassen, at indsamlede powerups, tilgængelig ammonition, liv, navn, position og hastighed, samt en del mere er tilgængeligt. Gameobjektet benytter blot en given spillers hastighed og `collision` metode til f.eks. at styre animationen af et hop, og indsamling af powerups. Det er også spiller-objektet, der styrer det grafiske omkring spilleren, så både modellen og det visuelle er samlet herfra.

5.6.2 Projektiler

Til vores projektiler har vi lavet klasse som hedder `Castable`, se Figur 3. Den opbevarer al nødvendigt data og nødvendige funktioner, som et projektil skal have. Dette er gjort for at for at kunne behandle alle projektiler ens og tilføje flere forskellige slags projektiler uden at ændre for meget koden i vores primære controller `GameObject`. Dette gør det også nemmere for os at uddele arbejdsopgaver, da man kan udvikle nye projektiler uden nogen yderligere koordination, da de automatisk vil blive handlet af koden allerede implementeret i `GameObject`



Figur 3: Diagram af nedarvning fra Castable-klassen vist uden getter- og setter-metoder. (Igen er teksten i hver image constructor erstattet med den egentlige sti til billedet i programmet).

5.7 Lyd og Musik

For at implementere lydeffekter og musik i spillet har vi gjort brug af to klasser: `Sound` og `Music`. `Sound` indeholder to vigtige funktioner. Man kan oprette et `Sound`-objekt, hvorved lyden bliver genereret via JavaFX' `Media` og `MediaPlayer`. Laver man et objekt får man en masse kontrol over lyden. Det giver mulighed for, at starte, stoppe og pause lyden. Skal vi bruge en hurtig lydeffekt giver klassen også mulighed for dette gennem en static metode. Her bliver der oprettet et `AudioClip`, der er optimeret til at blive brugt til korte lydklip. Bruger man denne metode, mister man dog de førnævnte metoder til at starte og stoppe lyden som det passer en. Til at afspille musik gør vi derfor brug `Sound`-objekter. De lydfiler, der indeholder musik bliver derefter behandlet af `Music`-klassen. Denne klasse gør det muligt at have en liste over sange, der én efter én automatisk bliver afspillet. Den finder automatisk ud af, hvornår en sang er færdig med at blive afspillet og starter derefter den næste sang.

5.8 Hop

Som nævnt tidligere, bruger vi JavaFx' `Timeline` til at animere spillernes hop. Man kan skifte til hoppe-tilstand ved at trykke på 'J' på tastaturet. Herved fremkommer der en parabel mellem spilleren og musen. Denne viser, hvordan spilleren vil hoppe, såfremt de ønsker dette. Et hop bliver animeret på samme måde som et projektil, men med en anden starthastighed. Hoppet stopper, når spilleren kolliderer med et andet objekt. Under hoppet tjekker programmet også, om man er landet på et power-up, og i så faldt bliver spilleren tildelt det pågældende power-up.

5.9 Sprites

I vores spil har vi masser af sprites, som vi selv har fremstillet. Disse bliver lavet til et `Image` i JavaFX og vist via et `ImageView`. Disse sprites er billeder som vi har lavet i programmet "Piskel". Dette program er nemt at lave pixel art i, da man let kan farve hver eneste pixel, og vælge størrelsen af billedet, og så kan man bagefter eksportere dem som png-filer, som kan blive

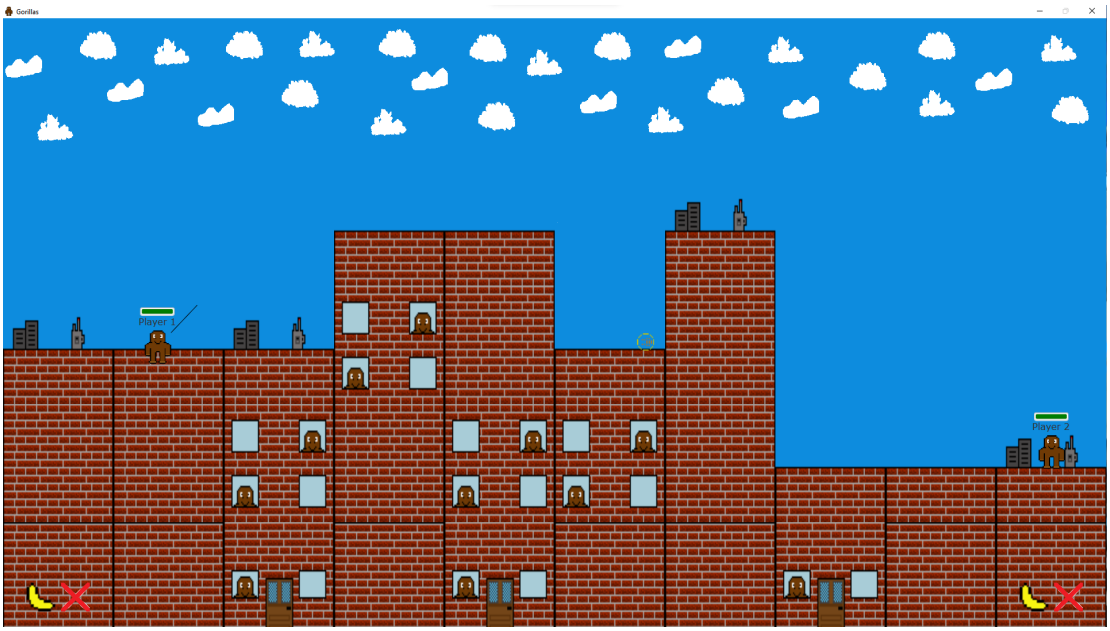
vist i spillet.
Vi har på denne måde lavet sprites til følgende:

- Bygninger
- Platformen
- Bananen og kokosnødden
- Gorillaernes forskellige stillinger
- Power Ups
- Baggrunden inde i selve spillet

Selvom størrelsen på billederne ikke passer helt med spillet, så kan man heldigvis nemt ændre størrelsen i programmet, så forholdene ser naturlige ud.

5.10 Mulige forbedringer

Selvom programmet i det store hele kører problemfrit, er der stadig nogle små fejl og mangler. Et problem, der nogle gange opstår, er at hoppe-animationen stopper for sent efter en kollision mellem spiller og bygning. Dette resulterer i, at spilleren kan havne med deres nederste halvdel inden i taget på en bygning.
Et andet problem er, at hoppelinjen ikke er helt så præcis som vi ønsker, og spilleren kan derfor ende længere fremme end tiltænkt ved et hop.
Desuden kunne nogle af vores hitboxes også være bedre, da vi har lavet de fleste af dem med forholdsvis simple former som cirkler og firkanter, som f.eks. ikke fungerer helt optimalt for bananen eller gorillaen, men godt til en kokosnød eller en bygning.



Figur 4: Eksempel på fejl med spiller-bygning-kollision, hvor 'Player1' er havnet halvt nede i en bygning

6 Testning

6.1 Prototype

Vi startede ud med at lave en prototype af spillet. Idéen med dette var, at vi kunne teste nogle forskellige måder at lave spillet på, og finde den metode, vi synes var mest effektiv. Her testede vi blandt andet, hvordan man kunne animere projektilerne. I prototypen testede vi en metode, hvor vi lavede et for-loop, der løb over antallet af pixels på x-aksen. Herved kunne vi opdatere y-værdien for hver x-position ved hjælp af formlen for det skrå kast. For at resten af spillet stadig kunne køre oprettede vi en ny thread, hver gang et projektil blev affyret. Vi fandt dog ud af, at dette var en ineffektiv metode til at animere et projektil og metoden var også fyldt med bugs. Nogle gange ville projektilet nemlig blive usynligt eller stoppe halvvejs igennem kastet. På denne måde fandt vi ud af, at det var bedre at bruge JavaFX' `Timeline` istedet. Det var også i prototypen, at vi testede brugen af aimlines, hvilket vi endte med at overføre til den rigtige version. I det hele taget blev prototypen brugt til at finde ud af, hvilke ting der er lette at implementere, og hvilke der er svære. På den måde kunne vi bedst prioritere vores tid og arbejdskraft, så vi kunne få det bedst mulige produkt.

6.2 Robusthed af programmet

Under udviklingen af spillet har vi løbende testet det. Efter stort set hver ny implementation, checkede vi om det virkede på den ønskede måde. Her har vi generelt været meget omhyggelige i forhold til alle slags input, som brugeren kunne komme med, hvilke edge-cases der kunne tænkes at være, samt overveje under hvilke omstændigheder koden kunne fejle. Her har vi både tænkt på håndtering af menuen, og i selve spillet, når f.eks. spillet skal kunne pauses på ethvert muligt tidspunkt. Denne rutinerede måde at tjekke vores implementationer på har medført, at programmet i sidste ende er meget robust overfor alskens input.

7 Projektstyring

Vi har i løbet af vores projekt brugt "Trello" til at styre arbejdsfordelingen. Her har vi opdelt alle opgaver i mindre opgaver, der er mere overskuelige. Vi har for eksempel inddelt programmeringsdelen i arbejdsopgaver, der omfatter de nødvendige klasser. På den måde har vi hver især kunnet arbejde på forskellige klasser og derved effektivisere arbejdsprocessen. Selvom vi har fordelt arbejdet ud så nogen har været ansvarlige for forskellige dele, så har vi altid kunne hjælpe hinanden med forskellige problemer, og har alle sammen bidraget til de forskellige ting. Når vi opdeler projektet på denne måde har vi haft behov for at kunne kommunikere med hinanden. Til dette har vi brugt Discord til at holde møder online, da vi, på grund af corona, ikke har kunnet mødes fysisk. Vi har haft møder 2-3 gange om ugen, hvor vi diskuterede, hvordan det stod til med hvert gruppemedlems arbejdsopgave. På møderne snakkede vi også om, hvem, der skulle stå for de næste dele af projektet. Vi har også haft et behov for at kunne implementere kode uafhængigt af hinanden og bagefter samle det til et stort projekt. Derfor har vi brugt GitHub til at dele programkoden med hinanden. Det har hjulpet med at forhindre, at alle ikke sidder og arbejder med det samme. Vi har i gruppen ikke haft nogen konflikter, men corona har gjort arbejdet sværere eftersom vi ikke kunne mødes, men blev nødt til at mødes online. Dette har dog ikke været noget stort problem.

8 Konklusion

I løbet af dette projekt har vi haft mange idéer og udviklet såvel på spillet, som vores som vores egne programmeringsegenskaber. Vi startede med at lægge en plan for vores ambitioner

og hvilke tilføjelser vi ønskede til grundspillet. Disse er beskrevet tidligere, og vi er glade for at have fuldført stort set dem alle.

Igennem projektet er vores program også blevet bedre fra vores første prototype til den simple version og helt hen til AdvancedGorillas, hvor vi har fået tilføjet mange spændende udvidelser som bl.a. forhindringer, forskellige ting at kaste med, mange sprites og Power-Ups. Selvom vi også havde flere ting som vi godt kunne tænke os at have haft med, såsom at bygningerne går i stykker, men det er svært da det kræver mange sprites for bygningernes forskellige tilstand. Derudover kunne vi også godt kunne have tænkt os at man skulle kunne gemme i spillet, men disse udvidelser har været lavere i prioritet end de andre, da vi fokuserede mere på et interessant spil og ikke havde tid nok til det hele. Vi har designet vores spil, så det følger MVC-konceptet, da konceptet passer godt til vores spil, og den måde brugeren interagerer med programmet på. I vores spil er der nogle af vores nuværende ting som vi kunne have forbedret bl.a. vores hitboxes så de passer bedre til formen på vores sprites, samt hoppe-funktionen som stadig har nogle småfejl. Vi har ikke lavet nogle documenterede tests, hvor vi testede at forskellige input gav de forventede resultater, men har igennem hele processen fundet og rettet små bugs ved bare at spille spillet.
Vi har i gruppen haft et godt samarbejde trods corona, som har gjort, at vi var nødt til at holde online møde og vi har desuden brugt "Trello" til at hjælpe os med at styre projektet. Trods corona har vi alle dog syntes, at det har været interessant og spændende at lave programmet og arbejde sammen i projektet.

Vi har gjort meget ud af den visuelle del af spillet. Vores program indeholder både en visuelt appellerende menu, håndlavede tegninger af objekter, animationer af kaste-objekterne samt af gorillaerne der kaster, og en bevægende baggrund.
Den vigtigste visuelle del i forhold til spil-oplevelsen er dog den sigtelinje, der tegnes ud fra en gorilla. Vi har valgt at bruge denne linje, frem for den oprindelige måde at skrive vinkel og hastighed i kommandolinjen, fordi vi vurderede at det giver en bedre bruger-oplevelse. Det er kluntet at skrive i kommandolinjen, og det er nemmere at sigte, når man kan bruge musen og en sigtelinje. Dog er det blevet sværere at vælge den korrekte hastighed, men dette er faktisk også en god ting, da det tilføjer et meget behøvet element af udfordring til spillet. Linjen er derfor en vigtig tilføjelse til spillet.